Introduction to Computer Organization – Fall 2023

Lab 8

Due: @11:55 PM, Wed November 1st

The following assignment is intended to be completed during your assigned lab period. One member of your group must submit the assignment to Gradescope by the posted deadline and indicate your group members when submitting the assignment. **Each group member must be present during the scheduled lab period in order to receive credit.**

Group names and uniqnames

Pod Member Name	**Uniqname**
Yuhan Zhang	zyuhan
Yuzhen chen	yuzhench

For each of the following problems, one person should act as the "scribe" and log the discussions of the group. You should rotate who is the scribe for each problem and indicate in the given space.

For this lab, please also time how long it takes your group to finish the problem.

Problem 1: ChatGPT *[10 points]*

Scribe: **yuzhen chen**

The following question was posed to ChatGPT: "Why is avoidance bad for solving data hazards in a pipelined processor?"

An abbreviated response is given below:

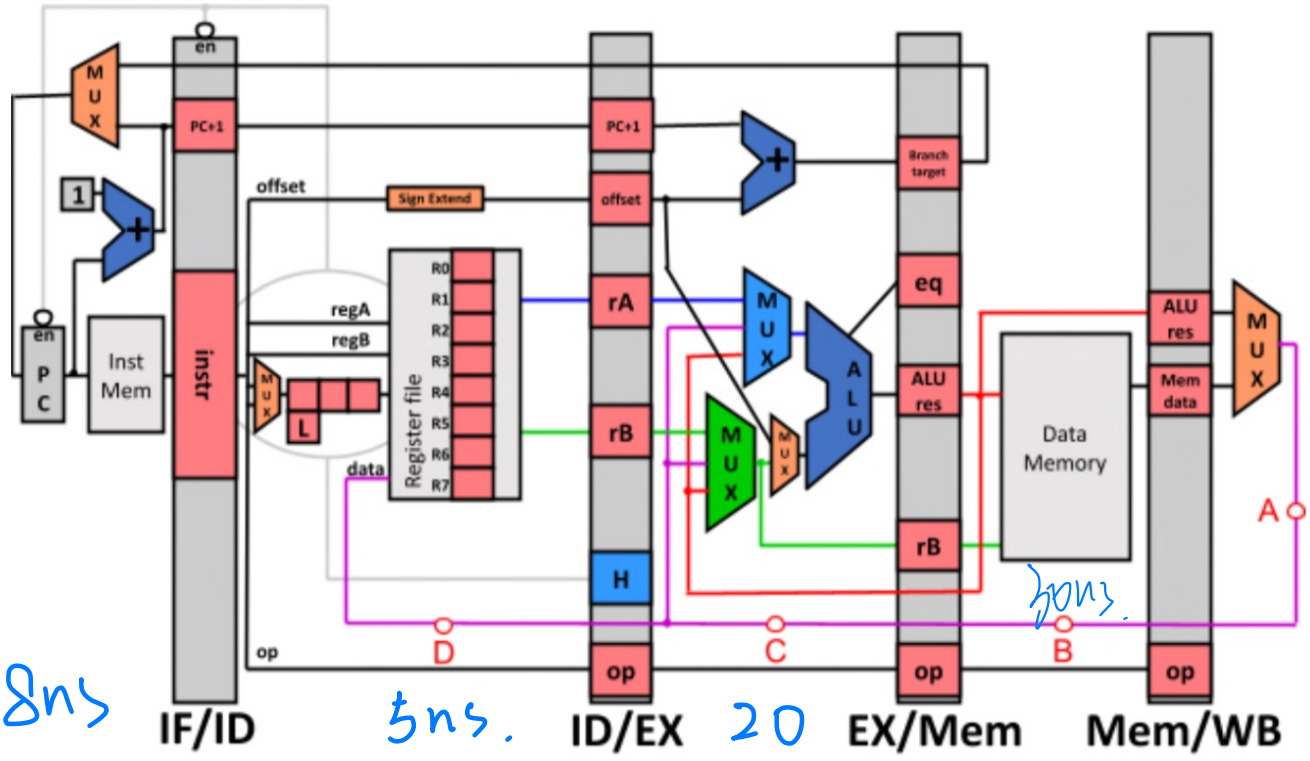
> *Avoidance, or stall, is a technique used in pipelined processors to mitigate data hazards. Data hazards occur when a pipeline stage requires data that is not yet available due to a dependency with a previous instruction. There are different approaches to handling data hazards, including forwarding and stalling.*
>
> *While avoidance or stalling can temporarily resolve data hazards, it is generally considered undesirable for several reasons:*
> *Performance impact: Stalling the pipeline introduces bubbles, or pipeline stalls, where no useful work is being performed.*
>
> *Increased latency: Stalling introduces additional clock cycles or delays in the pipeline.*
>
> *Wasted resources: Stalling requires additional hardware resources to manage and detect hazards, as well as to handle the control signals for stalling. These extra resources increase the complexity and cost of the processor design.*
>
> *Limited effectiveness: While stalling can resolve some data hazards, it does not eliminate them entirely. In cases where there are multiple dependent instructions in the pipeline, stalling for every hazard can result in a significant number of stalls and reduce the benefits of pipelining.*

This is a pretty good answer, although it seems to have conflated "avoidance" and "detect-and-stall". Explain what the difference between detect-and-stall and avoidance is. What is a main disadvantage of avoidance compared to detect-and-stall? Do any of the above arguments not apply to avoidance? Your response should be about 1 paragraph.

Answer:
The difference between detect-and-stall and avoidance is for avoidance, the people who write the code take the responsibility to avoid the data hazard, if the next instruction need to read the register that will be uodated by the previous commands, then the writer need to add noop between them in order to let the instruction read in the correct data. However, in this case, we don’t need to change the hardware. For Detect-and-stall: We need to design new hardware resources to first detect the data hazard, then freeze the pc+1 and replace the next instruction with noop untile the correct data is updated in the register file or the data memory.

The main disadvantage of avoidance compared to detect-and-stall is if the pipeline structure changes, then the number of the noop will also change, so the old program (legacy code) may not run correctly on the new implementations. The arguments below does not apply to avoidance: "*Wasted resources: Stalling requires additional hardware resources to manage and detect hazards*". For avoidance, we don't need to add additional hardware resources.

Problem 2: Pipeline Design Modification *[22 points]*



Pipeline Datapath for Reference

Consider the pipeline from lecture (above) with times below, using **detect and forward** for data hazards and **speculate taken and squash** for control hazards. The current cycle time is 30ns.

- Data Memory Read/Write: 30 ns
- Instruction Memory Read: 8 ns
- ALU: 20 ns
- Register file read/write: 5 ns
- Backwards wire delay: 2 ns per stage *
- All other wires: 0 ns

- All other delays: 0 ns

* Wires that go backwards across pipeline stages (eg. mux in WB to the Register File, pipeline registers to muxes in EX) have a delay of **2 ns per stage**. Thus the delay from point **A** (the black circle labeled A in WB) to **B** (in MEM) is 2 ns, and the delay from **A** to **D** is 6 ns. This also affects forwarded values: for example, **A** to **C** is 4ns.

The current pipeline is: IF → ID → EX → MEM → WB

Suppose we have the option to add new stages to the pipeline and *evenly* split memory transactions and ALU operations into up to 4 stages each. E.g. we could split the ALU into two stages to create a new pipeline:

IF → ID → EX1 → EX2 → MEM1 → WB

In this new pipeline, the 20 ns ALU operation is divided into two 10 ns operations, one in EX1 and one in EX2. However, the delay from point **A** (in WB) to **C** (in EX) is now 6 ns, so the propagation delay in EX1 is 16ns. And the delay from point **A** (in WB) to **D** (in ID) is now 8ns, so the WB stage delay is 13ns. (Data can only be forwarded from MEM1 and WB, data is always forwarded to EX1, and register reads and writes are concurrent due to internal forwarding).

a) The below pipeline design has the optimal clock period given the above constraints. What is this clock period? Show your work. *[8 points]*

Pipeline: IF → ID → EX1 → EX2 → EX3 → MEM1 → MEM2 → WB

Clock period: 17 ns

b) This pipeline, though it has more stages, has a higher clock period. What is this clock period? *[6 points]*

Pipeline: IF → ID → EX1 → EX2 → EX3 → EX4 → MEM1 → MEM2 → MEM3 → WB

Clock period: 21 ns

c) For the optimal design in part a), complete the table to indicate how many *stalls* are needed for register dependencies that cannot be completely solved by forwarding. For example, in the original pipeline, a dependent instruction immediately after an LW would require 1 stall. *[8 pts]*

Source Instr. Type	Destination Distance	# Stalls
Ex for OG design: LW	*1 (Immediately after)*	*1*
Add/Nor	1	2
Add/Nor	2	1
LW	1	4
LW	2	3
LW	3	2

LW	4	1

Problem 3: P3 Tests *[18 points, Autograded]*

With project 3 right around the corner, let's get ahead and find some of those buggy instructor solutions!

- For this problem, your group will be submitting (to autograder.io **as a group**) 2 versions of assembly code per test case.
- Once you have written test cases that expose the bugs, you must modify the assembly code to contain the ***minimal*** number of noops necessary to avoid register data hazards in the project 3 checkpoint pipeline. Then, you will also submit this code to the autograder for full credit.
- For every bug that is caught, you will receive 6 points. So for full credit, you need to catch 3 bugs with your tests, and provide the correct hazard-free assembly for each.
- In addition to the constraints listed in the project, each LC2K program you write must be limited to **10 lines** or fewer (including .fill directives) and must execute no more than 25 cycles.
- Tests should assemble properly, and should not contain beq, jalr, or self-modifying code. That is, there should be no control hazards.
- Each submission will be limited to 3 test cases, but fewer may be needed.
- Each modified file name must be the same as the original, with the extension changed to .nohaz.
- You are free to resubmit tests you wrote for previous projects, and you may submit these for Project 3 as well without honor code penalty (as long as they were written by members within your group or provided in course materials).
- You are encouraged to use (and submit) these test cases if you are still working on P3. A great strategy is to run these hazard-free test cases on your pipeline simulator and "diff" your output with the correct output from your checkpoint-version simulator any time you make a change.

Example.as:	Example.nohaz:

`add 0 0 1`
`add 1 0 2`
`halt`
	`add 0 0 1`
	`noop`
	`noop`
	`noop`
	`add 1 0 2`
	`halt`